Начиная с этого занятия, мы с вами рассмотрим возможность переопределения стандартных операций для объектов классов. И начнем с операции сложения.

О чем здесь речь? Давайте предположим, что у нас есть класс для представления времени:

```cpp
class TimeDay {
    unsigned long time {0};

public:
    TimeDay() : time (0)
        { }
    TimeDay(unsigned char hs, unsigned char ms, unsigned char ss, unsigned int
ds = 0)
        : time(ss + ms * 60 + hs * 3600 + ds * 86400)
        { }

    void get_time(unsigned int& days, unsigned char& hours, unsigned char& mins,
unsigned char& secs) const
    {
        secs = time % 60;
        mins = time/60 % 60;
        hours = time/3600 % 24;
        days = time/86400;
    }
};
```

Время здесь раскладывается по дням, часам, минутам и секундам. В самом классе TimeDay хранится одно число time – время в секундах. Далее, в функции main мы можем использовать этот класс, например, так:

```cpp
int main()
{
    TimeDay t1(10, 45, 13), t2(4, 11, 50);

    unsigned d;
    unsigned char h, m ,s;

    t1.get_time(d, h, m, s);
    printf("%u: %02u:%02u:%02u", d, h, m, s);
```

```
    return 0;
}
```

Предположим теперь, что нам хотелось бы иметь возможность складывать объекты этого класса:

```language-cpp
TimeDay res = t1 + t2;
```

Но операция сложения для них не определена и компилятор выдаст ошибку. Однако мы можем добавить эту возможности и задать порядок выполнения операции сложения для наших объектов. Для этого в любом классе можно объявить специального вида метод, имя которого состоит из ключевого слова operator с символом переопределяемой операции:

```language-cpp
class TimeDay {
    unsigned long time {0};

public:
...
    const TimeDay& operator + (const TimeDay& right) const
    {
    }
};
```

Я его записал с одним возвращаемым значением в виде константной ссылки на объект класса TimeDay, и одним параметром right тоже в виде константной ссылки. За что отвечает этот параметр и почему возвращается константная ссылка? Смотрите, когда выполняется сложение:

```language-cpp
TimeDay res = t1 + t2;
```

то компилятор автоматически подставит вместо сложения вызов нашего метода. При этом текущий объект, из которого был сделан вызов – это объект t1, в параметр right подставляется объект t2, а возвращаемое значение копируется в переменную res.

На самом деле это же сложение можно записать и с явным вызовом метода-оператора:

```language-cpp
TimeDay res = t1.operator+(t2);
```

То есть, operator+ - это тоже метод, но который используется компилятором при вычислении операции сложения. Там, где она встречается

для объектов класса TimeDay, будет использоваться метод operator+.

Теперь, почему в качестве параметра и возвращаемого типа используется константная ссылка? Конечно, можно было бы записать этот оператор и без ссылок. Результат от этого не изменился бы. **Но тогда дополнительно выполнялась бы операция копирования объекта класса TimeDay в параметр right, и также происходило копирование при возврате значения. Это лишние операции, поэтому часто при переопределении операций используют ссылки.** То же относится и к ключевому слову const, которое не обязано быть прописанным. Опять же – **это правило хорошего тона** при вызове метода, который гарантирует неизменность передаваемого объекта, не более того.

Давайте пропишем реализацию метода операции сложения:

```language-cpp
class TimeDay {
    unsigned long time {0};

public:
...
    TimeDay(unsigned long tm)
        : time (tm)
        { }

...
    TimeDay operator + (const TimeDay& right)
    {
        return TimeDay(this->time + right.time);
    }
};
```

Для удобства я добавил еще один конструктор в класс с одним обязательным параметром. Напомню, что такой конструктор называется конструктором преобразования. В частности, он тип long переводит в объект TimeDay. **Кроме того, сделал возврат копии объекта TimeDay вместо ссылки, так как созданный в методе временный объект перестает существовать при завершении этого метода. И, если бы была ссылка, то она вела бы на не существующий объект**. Это тоже следует учитывать, когда мы используем ссылки в параметрах или возвращаемых значениях. Нужно быть уверенным в том, что объекты существуют необходимое нам время.

Благодаря добавлению конструктора преобразования, мы можем им воспользоваться следующим образом:

```language-cpp
TimeDay res = t1 + 10;
```

Здесь 10 в операции сложения воспринимается компилятором, как целочисленный литерал типа int. **Затем, с помощью конструктора преобразования число 10 (время в секундах) трансформируется в объект класса TimeDay и вызывается метод operator+**. Результат копируется в переменную res. Напомню, что копирование встроено для объектов классов и структур и для этого вызывается конструктор копирования по умолчанию, который побайтно копирует данные из одного объекта в другой. В данном случае, нас это устраивает.

Так же, учитывая наличие конструктора преобразования, метод operator+ теперь может возвращать обычное число типа long:

```language-cpp
   unsigned long operator + (const TimeDay& right) const
    {
        return this->time + right.time;
    }
```

Тогда в момент инициализации объекта res, это число сначала будет преобразовано во временный объект TimeDay, а затем, с ним будет вызван конструктор копирования.

Однако, обратите внимание, если операцию сложения наоборот:

```language-cpp
TimeDay res = 10 + t1;
```

то возникнет ошибка на этапе компиляции. Так как метод operator+ должен вызываться через объект int:

```language-cpp
TimeDay res = 10.operator+(t1);
```

А такого метода у объекта int просто не существует. Но мы все же можем выйти из этой ситуации, если определим операцию сложения в виде функции, а не метода. Да, так тоже можно делать. В нашем случае ее можно было бы записать так:

```language-cpp
unsigned long operator + (const TimeDay& left, const TimeDay& right)
{
    return left.get_time() + right.get_time();
}
```

И добавить метод get_time в класс TimeDay:

```cpp
unsigned long get_time() const { return time; }
```

Компилятор автоматически выберет подходящий метод для операции сложения и подставит его вызов при выполнении этой операции. **Однако может возникнуть конфликт таких операторов, если объявлен метод и в классе и в виде отдельной функции**. Если мы поменяем операнды в операции сложения:

```cpp
TimeDay res = t1 + 10;
```

то возникает неопределенность: какой из двух подходов использовать? Поэтому **операции следует переопределять либо методами, либо функциями**.

Конечно, здесь может возникнуть вопрос, зачем вообще разрешили делать переопределение на уровне функций, или, наоборот, на уровне методов? Были бы только методы и такой неопределенности не возникало бы? Причина такого разнообразия в том, что иногда требуется определить операции для объектов уже существующих классов, которые мы не можем редактировать напрямую. И функции здесь единственная возможность для переопределения стандартных операций.